TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

SYSTÉM PRO PRŮBĚŽNOU INTEGRACI SOFTWARU

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika

Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Vojtěch Tůma**

Vedoucí práce: Mgr. Jiří Vraný, Ph.D.

Liberec 2014

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

SYSTEM FOR SOFTWARE CONTINUOUS INTEGRATION

Diploma thesis

Study programme: N2612 – Electrotechnics and informatics

Study branch: 1802T007 – Information technology

Author: **Bc. Vojtěch Tůma**

Supervisor: Mgr. Jiří Vraný, Ph.D.

Liberec 2014

Tento list nahraďte

originálem zadání.

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum:

Podpis:

Abstrakt

Práce se zabývá problematikou průběžné integrace softwaru a obecně vývoje softwaru v týmu. V úvodu práce jsou podrobně popsány všechny nástroje průběžné integrace. Tedy verzovací systémy, testování softwaru a sestavovací skripty. V druhé části práce je popsán návrh a implementace vlastního systému pro průběžnou integraci. Popis systému se skládá ze 3 částí. První část popisuje grafické rozhraní pro ovládání celého systému. Druhá část se zaměřuje na popis vlastního řešení pro vytváření sestavovacích skriptů. Poslední část stručně popisuje komponentu autorizující uživatele skrze protokol SSH ke vzdáleným repozitářům. Cílem práce je, aby čtenář porozuměl a naučil se používat autorem navržený systém pro průběžnou integraci softwaru.

Klíčová slova: průběžná integrace, verzovací systémy, testování softwaru, sestavovací skripty

Abstract

Work is focused on prolematics of continuos integration and team software development. Introduction describes all tools of continuos integration. Means version control system, software testing and build scripts. In the second part of work there is described own implementation of system for continuous integration. Description of system is divided in to three parts. Graphical interface for controling whole system id described in the first part. Second part describes own implementation of creating buildscripts. The last part describes component which authorize users to access remote repositories throw SSH protocol. Mail goal of work is to explaim how created system for continuos integration works and explaim to reader how to use it.

Keywords: continuos integration, version control systems, software testing, build scripts

Poděkování

Chtěl bych poděkovat Martinu Takáčovi za pomoc při vytváření systému. Dále bych chtěl poděkovat vedoucímu práce Mgr. Jiřímu Vranému Ph.D., který se mé práce ujal. V neposlední řadě chci poděkovat svým blízkým, kteří mě podporovali během studia.

Obsah

Seznam obrázků

Seznam zkratek

CI	Continuous integration(Průběžná integrace)
FTP	File Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
SVN	Subversion
XML	eXtensible Markup Language
IDE	Integrated Development Environment
SSH	Secure Shell(protokol pro zabezpečenou komunikaci)

1 Úvod

Pokud se ohlédneme do nejmladších dob naší civilizace zjistíme, že již pračlověk používal nástroje, aby si ulehčil svoji práci. Ve středověku si každý kovář vyráběl nástroje, pomocí kterých mohl vyrábět rychleji dokonalejší výrobky. Dokázal vyprodukovat více kvalitních výrobků a uspokojit lépe svoje zákazníky. Vytváření nástrojů, které ulehčí práci, se zdá tedy být výhodné. Díky mnohým z těchto nástrojů, dosáhla naše civilizace velké technické úrovně.

Jedna z dnešních velmi důležitých profesí je programování. I zde lidé hledají nástroje, pomocí kterých by si svou práci mohli usnadnit. Na rozdíl od povolání kováře se ovšem jedná o velmi mladou disciplínu. V posledních desetiletích však získává na rozmachu. S příchodem prvního elektronkového počítače ENIAC, který byl dokončen v roce 1946, se mnohé změnilo. V této době existovala pouze velmi omezená množina vyjímečně nadaných jedinců, kteří dokázali naprogramovat funkční aplikaci. Pokud aplikaci vyvíjel jeden či dva lidé, nebyl problém si práci rozdělit.

Dnes se programování přiblížilo mnohem větší skupině lidí. Nyní může na jednom projektu pracovat i několik desítek programátorů. Pokud by každý pracoval na jiné části kódu, nenastal by žádný problém a s trochou nadsázky by se projekt mohl vyvíjet bez omezení. Může se však stát, že dva různí lidé pracující na odlišných částech aplikace. Budou potřebovat upravit tu samou knihovnu, protože v ní byla chyba. Zde může vznikout konflikt, jestliže oba neopravili knihovnu naprosto steným

způsobem. Pokud se jedná o přidání jednoho řádku, tak problém nenastává. Pokud se však změní několik desítek řádků v jedné knihovně, je prakticky nemožné práci obou správně propojit a ověřit správnou funkčnost.

Tato práce se zabývá problematikou práce více lidí na jednom projektu. Jak vytvořit možnost přidat do existujícího projektu novou funkcionalitu aniž by došlo k rozbití té stávající.

2 Průběžná integrace

Pro vysvětlení pojmu průběžná integrace (CI) je nejlepší si uvést příklad. Mějme tedy za úkol vytvořit novou vlastnost do již existujícího webového projektu. Nejprve si uvedeme příklad postupu, kde bude použitý systém průběžné integrace nebo jeho částí a posléze příklad, kdy nebude využito žádné části průběžné integrace.

Pokud nepoužijeme žádné z částí průběžné integrace, bude náš postup většině programátorů velmi dobře známý. Otevřeme souborový manažer a zkopírujeme si existující kód projektu na svůj počítač. Následně provedeme úpravy na různých knihovnách, do kterých přidáme nové vlastnosti pro náš projekt. Otevřeme webový prohlížeč a vyzkoušíme všechny nové vlastnosti na lokálním počítači. Pokud vše funguje správně, nakopírujeme pomocí protokolu FTP(S) nebo protokolu SSH na server upravené soubory. Nyní znovu zkontrolujeme ve webovém prohlížeči, zda všechny naše nové vlastnosti fungují. Je–li součástí nových vlastností i aktualizace databázového schématu, musíme ručně databázi upravit. Může zde nastat několik problémů. Může to být zapomenutý upravený soubor, nezkontrolování všech částí, kterých se aktualizace týkala nebo nesmazání vyrovnávací paměti, kde jsou stále uložená stará data.

Nyní použijeme nástroje pro průběžnou integraci, které pomáhají zamezit právě těmto chybám. V tomto případě budeme postupovat analogicky, ale budeme využívat nástroje, které nám práci ulehčí. Samotné nástroje budou popsány v dalších kapi-

tolách. Pro zkopírování aktuálního kódu projektu na vlastní počítač využijeme verzovací systém, který vytvoří pracovní kopii projektu. Následně provedeme úpravy v kódu. Pokud náš projekt obsahuje jednotkové či jiné testy, nástroje tyto testy automaticky spustí před odesláním nového kódu na server. Pokud některý z testů bude hlásit chybný výsledek víme, že nová vlastnost narušila již existující kód nebo nefunguje správně. Verzovací systém následně odešle na server všechny soubory, které byly upraveny. Pokud aktualizace upravuje databázové schéma, tak i tyto změny jsou provedeny automaticky. Máme tedy mnohem větší šanci úspěchu, že naše nová vlastnost bude správně spuštěna.

Tímto by se dal velice krátce vysvětlit pojem průběžná integrace. Nyní si popíšeme jednotlivé části, které by měl systém pro průběžnou integraci podporovat.

2.1 Verzovací systém

Mějme situaci, kdy dva různí lidé mají za úkol do existujícího projektu přidat svůj vlastní obsah. Tento projekt může být ku příkladu kniha. Nemusí se vždy jednat o software. Každý si tedy na začátku pořídí kopii aktuální verze. První z nich dokončí práci dříve a na několik různých míst v knize přidá náhodné množství nových odstavců a upraví i části původního textu, protože obsahoval chyby. Druhý člověk dokončí práci jen o den později s tím, že připsal jeden odstavec na konec knihy.

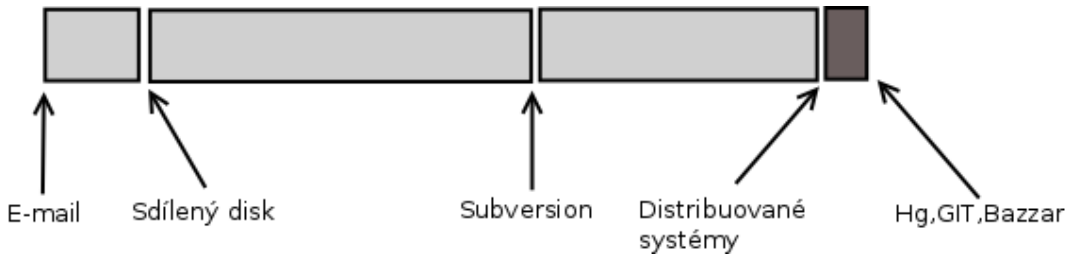
Nejjednodušší řešení je vzít práci člověka, který vytvořil více textu a jehož práce je více rozprostřena v celé knize a k ní připojit nový odstavec na konec, který vytvořil druhý člověk. Nyní žádný problém nenastává, protože oba se dokázali dohodnout, co kdo udělal. Bohužel druhý z nich zapomněl říci, že opravil na začátku knihy úvod, bez kterého odstavec na konci knihy nemá smysl. Knihu vydají a tento nedostatek

objeví až poté, co se začne kniha prodávat. Čtenáři si povšimnou, že poslední kapitola nedává smysl. Vydavatel knihy tedy zjistí, že je v knize chyba a vydá nový výtisk. Bohužel, už vytiskl několik stovek výtisků, které obsahují chybu. Zkusí tedy nalézt lepší proces kontrolování práce.

Kniha se nyní opět nachází v aktuálním stavu (všichni mají stejnou kopii). Editorů knihy začne přibývat a zjistí, že je lepší si každý den poslat seznam změn, které každý z nich provedl. Tento systém je už lépe udržitelný, protože jednou za den si všichni do svého rozpracovaného dokumentu nakopírují nové části. Dalším pokrokem je využívání sdíleného disku, kdy existuje jeden soubor na sdíleném úložišti a každý do něj může naráz přispívat. Jelikož je aktuální práce všech přispěvatelů na jednom místě, má smysl dokument v pravidelných časových intervalech zálohovat. Tím získáme historii dokumentu a můžeme se zpětně podívat ,v jakém stavu byl dokument minulý týden.

Pokud k dosaženému pokroku přidáme možnost kontrolovat, kdo jaké změny udělal a v jaký čas, tak se dostáváme na úroveň nejzákladnějšího verzovacího systému. Takovým systémem může být například Subversion[1] (označován zkratkou SVN), který za nás provádí základní operace jako překopírování aktuální verze ze serveru na lokální počítač a zároveň i nahrání upravené verze zpět na server. Ke každé změně je možné přidat i text, který bude popisovat provedenou změnu. Pokud uděláme změnu a nahrajeme ji na server, ostatní uživatelé si ji mohou stáhnout k sobě na lokální počítač a pokud se nejedná o kolizní změnu (dva přispěvatelé naráz upravili stejný řádek) bude změna automaticky zakomponována. Nyní máme již velice silný nástroj, který dokáže zajistit automatické připojování změn mezi přispěvateli a zaznamenávat, kdo co upravil. Pořád se ovšem jedná o chytřejší sdílený disk, který má vše důležité uložené na jednom místě, proto se tento systém nazývá centralizovaný. Vše je uloženo na jednom centralizovaném úložišti, ke kterému musíme mít stálé připojení, pokud chceme projekt upravovat.

Všechny tyto nedostatky odstraňují distribuované verzovací systémy[2]. Přinášejí možnost pracovat bez přímého připojení k centralizovanému úložišti. Změny se mohou hromadit na lokálním počítači a odeslat je na centrální úložiště až v době, kdy je možnost se k němu připojit. Můžeme tedy udělat několik změn a ty například na konci týdne odeslat, což ovšem není dobrý zvyk, jak bude popsáno dále. Největší výhodou je distribuce změn mezi všemi přispěvateli. Každá instance (kopie, klon) má v sobě uložené změny, které provedli ostatní přispěvatelé. Je možné tedy odvážně říci, že nemá smysl vytvářet speciální zálohy na centrálním úložišti, protože na všech počítačích je záloha kompletně uložená. Mezi nejznámější zástupce distribuovaných systémů patří GIT, Mercurial(Hg) a Bazzar(Bzr).

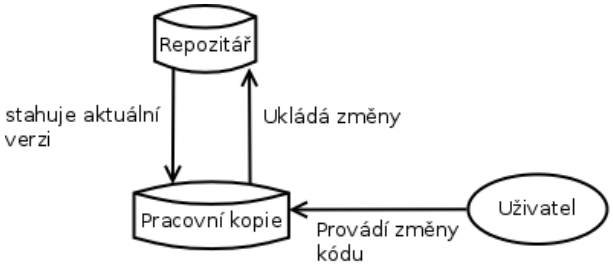


Obrázek 2.1: Velikost přínosů při pokroku

2.1.1 Repozitář

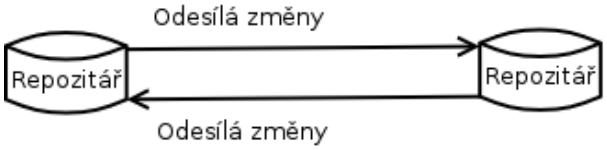
Nyní knihu z prvního příkladu nahradíme za aplikaci, která obsahuje několik souborů. Soubory musí být někde uloženy. U verzovacích systémů se toto místo nazývá repozitář. Jedná se o adresář, kde jsou ve speciálním formátu uloženy všechny soubory, které jsou verzované (spravované verzovacím systémem). Správně není možné do repozitáře přistoupit a ručně změnit jeho obsah. Můžeme si ho představit jako balík dat, který obsahuje námi vložený obsah a doprovodná data, která popisují jednotlivé změny provedené na datech.

V případě, že nepracujeme v týmu a chceme pouze ukládat historii projektu, stačí nám lokální repozitář. Ten leží na lokálním počítači a zpravidla k němu má přístup pouze uživatel na daném počítači. Aby bylo možné data v repozitáři upravovat, musíme si vytvořit pracovní kopii repozitáře, ve které můžeme provádět změny, které opět následně nahrajeme do repozitáře. Ten změnu zaznamená a uloží si ji do své historie.



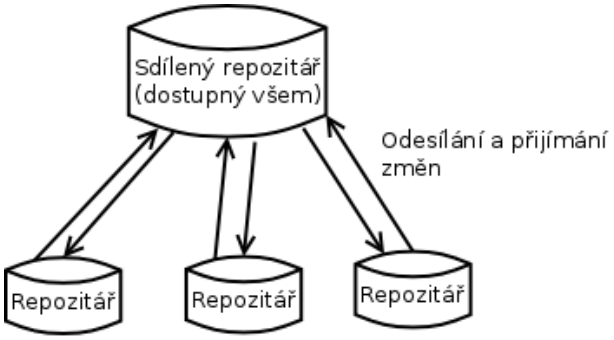
Obrázek 2.2: Repozitář a pracovní kopie

Práce v týmu ovšem vyžaduje tyto repozitáře synchronizovat. Repozitáře mezi sebou dokáží komunikovat a odesílat si mezi sebou změny. Je tedy možné propojit dva repozitáře, které si mezi sebou budou odesílat změny.



Obrázek 2.3: Synchronizace dvou repozitářů

Toto se ovšem stává nepraktické, když se zvyšuje počet repozitářů a vzniká těžce udržitelná infrastruktura. Proto se zavádí myšlenka sdíleného repozitáře, se kterým se všechny ostatní repozitáře aktualizují. Tento repozitář by měl být na veřejném místě, kam mají všichni uživatelé přístup, aby se s ním mohli co nejčastěji synchronizovat. V praxi se toto řešení ukazuje jako nejlepší.



Obrázek 2.4: Synchronizace dvou repozitářů

Repozitáře, které nejsou sdílené se nazývají lokální nebo uživatelské. Pro zjednodušení dále v práci budou tyto repozitáře komponovány jako součást uživatele a budeme uvažovat, že uživatel komunikuje se sdíleným repozitářem právě skrze svůj lokální repozitář.

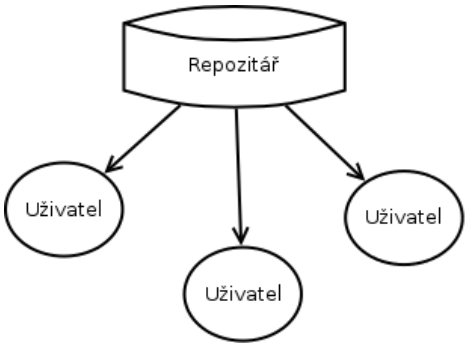
2.1.2 Životní cyklus verzování

Pokud se zaměříme na nejpokročilejší distribuované verzovací systémy, všechny mají velice podobný životní cyklus. Základem je vytvoření repozitáře, který je přístupný všem přispěvatelům. Pro firemní projekty je dobré ho umístit na sdílený disk a zpřístupnit ho požadovaným uživatelům. Není problém se však připojit ke vzdálenému repozitáři promocí protokolu SSH nebo HTTPS i vně firemní sítě. Většina distribuovaných verzovacích systémů má již připravené rozhraní, které implementaci této komunikaci zajišťuje. Existují dva možné scénáře při vytváření repozitáře:

- Obsah (kód) již existuje a repozitáře bude obsahovat nějaká data
- Vytvoříme prázdný repozitář, který následně naplníme obsahem

Na základě situace vytvoříme nový repozitář, který je prázdný nebo inicializujeme repozitář nad již existujícím kódem. V obou situacích se vlastně jedná o vy-

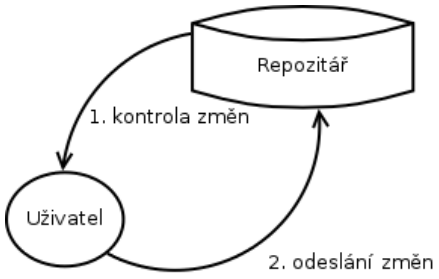
tvoření prázdného repozitáře, do kterého se následně nahrají data.



Obrázek 2.5: Rozkopírování repozitáře

Požadování uživatelé si následně vytvoří kopie sdíleného repozitáře.

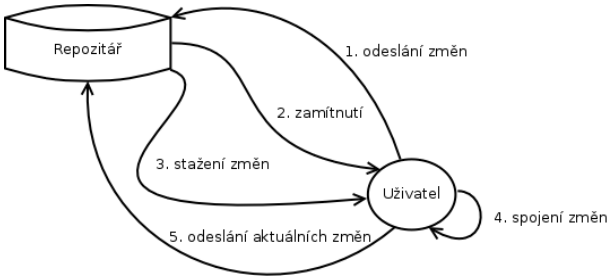
Nyní už má každý uživatel vlastní kopii a může vytvářet změny. Až uzná za vhodné, tak odešle seznam svých změn do sdíleného repozitáře. Nejlepší je posílat změny co nejčastěji, aby nedocházelo ke zbytečně velkému množství konfliktů. Nejdříve systém zkontroluje, zda nejsou nové změny ve sdíleném repozitáři od jiných uživatelů a pokud nejsou, tak změny odešle.



Obrázek 2.6: Odeslání změn do repozitáře

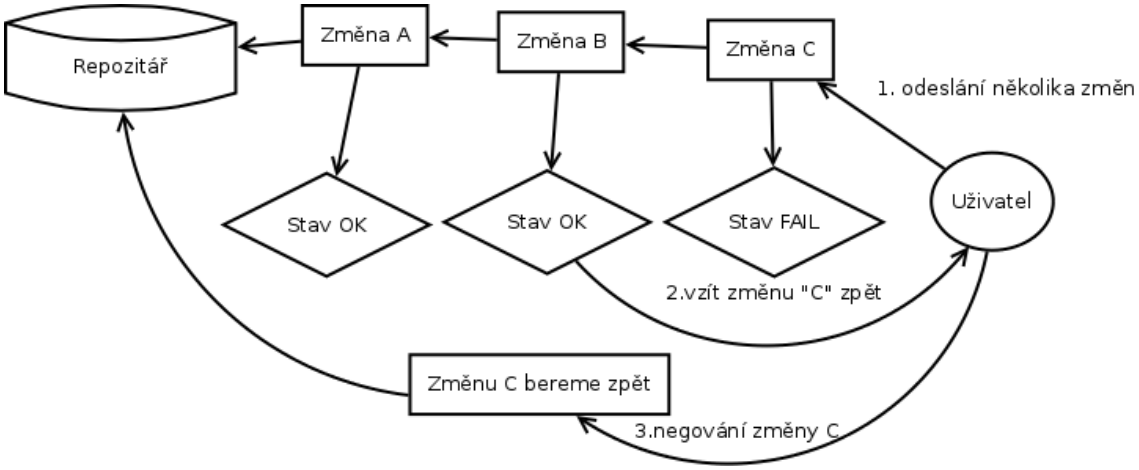
Pokud nějaký jiný uživatel udělal změnu, která ještě není v lokálním repozitáři zahrnuta, tak se nahrání do sdíleného repozitáře neprovede. Systém upozorní na vznik konfliktu a bude po nás požadovat stažení změn od ostatních uživatelů. Před odesláním musíme vlastní změny spojit se změnami, které provedli ostatní uživatelé

a výsledek následně odeslat do sdíleného repozitáře.



Obrázek 2.7: Přidání ostatních změn a odeslání

Jednou z posledních důležitých událostí je navrácení změn. Pokud nějaký uživatel udělá opravu, která rozbije funkčnost aplikace, je nutné se vrátit o jednu nebo několik verzí zpět. Jedná se vlastně o klasickou změnu v kódu s tím, že změníte kód do stavu, ve kterém již byl.

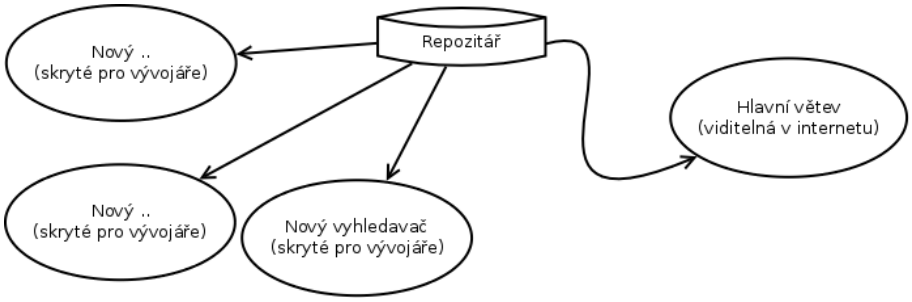


Obrázek 2.8: Navrácení změn

Jak znázorňuje obrázek 2.8, uživatel odeslal do sdíleného repozitáře 3 změny a každé změně odpovídá stav aplikace. Změna C dostala aplikaci do stavu, kdy nefunguje. Uživatel tedy chce vrátit aplikaci do stavu, kdy ještě fungovala. Vezme tedy změnu C zpět tím, že další změnou neguje vše co se v C změnilo.

2.1.3 Vývojové větve

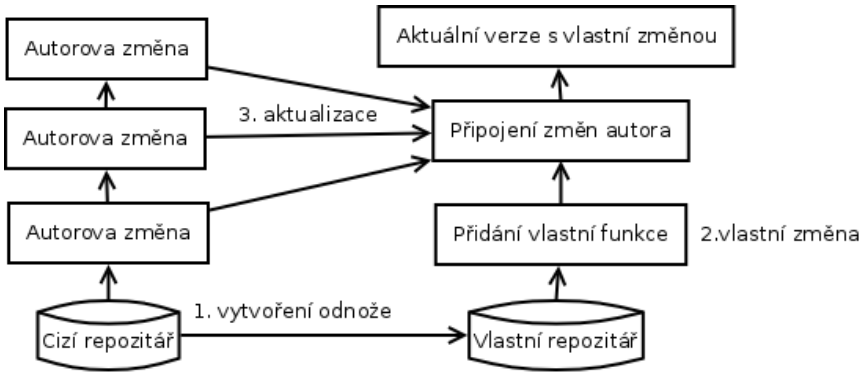
Jestliže verzovací systém spravuje například internetový vyhledávač stránek a uživatel provede úpravu, kterou odešle na sdílený repozitář, úprava se okamžitě projeví v ostré verzi, která je viditelna všem návštěvníkům vyhledavače na internetu. Pokud by si ovšem uživatel chtěl pouze vyzkoušet nový způsob vyhledávání nebo novou barvu pozadí, bylo by dobré mít skrytou kopii, kterou uvidí pouze on. Tento problém se dá vyřešit vytvořením dalšího repozitáře, který nebude umístěn viditelně na internetu. Ovšem pak musíme ručně řešit synchronizaci mezi repozitáři a tím se dostáváme na úroveň dokumentu na sdíleném disku.



Obrázek 2.9: Viditelnost vývojových větví

Proto verzovací systémy přicházejí s možností zakládat vývojové větve. Umožňují mít hlavní větev, která je například viditelná všem uživatelům internetu, ale k ostatním větvím se dostane pouze omezená skupina lidí.

Dalším využitím vývojových verzí je vytvoření odnože cizího software. Jestliže vámi oblíbené knihovně schází právě jedna pro vás klíčová funkce, kterou víte jak doprogramovat, ale nechcete celou knihovnu vytvářet znovu, je pro vás nejlepší možností vytvořit odnož již existující knihovny. Jedná se o zkopírování poslední verze knihovny do vlastního repozitáře, kde knihovně doplníte potřebné funkce. Pokud autor knihovny vydá aktualizace, které pokud nekolidují s vašimi změnami, můžete si aktualizaci přidat i do vlastní knihovny.



Obrázek 2.10: Vytvoření odnože

2.1.4 Zachytné háky

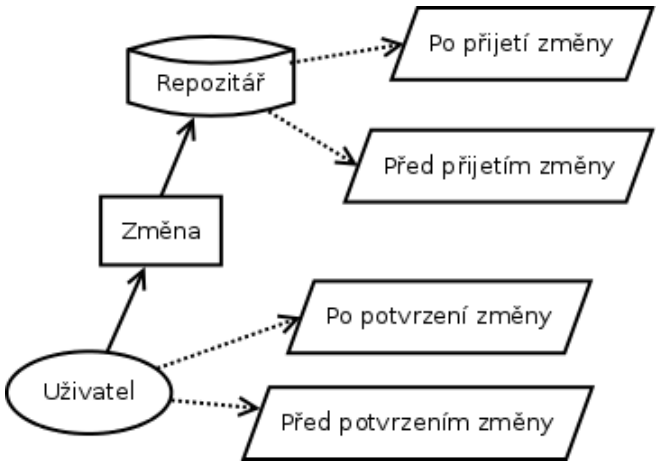
Pokud jeden z přispěvatelů provede nahrání svých změn do sdíleného repozitáře, bylo by dobré na to upozornit ostatní. Nebo spustit automatickou sadu testů, která ověří, že se změnou nic nerozbilo.

Toto je možné právě pomocí záchytných háků (hooks). Verzovací systémy poskytují několik háků, na které se dají „pověsit“ různé skripty. Většinou lze zvolit, zda chceme skript spustit před nebo po dané události. Jednotlivé háky jsou rozděleny na lokální a serverové.

Lokální háky (v uživatelksých repozitářích) se většinou používají na syntaktickou kontrolu změn, které odesíláme. Můžeme zde kontrolovat, zda změny neobsahují zakomentovaný kód nebo ladící informace, které by měly zůstat pouze na lokálním počítači uživatele.

Serverové háky (ve sdíleném repozitáři) naopak nacházejí využití při spouštění testů, sestavování či publikování aktuální verze pro veřejnost. Jestliže například selže

testování aplikace, budou všichni uživatelé upozornění e-mailem. Záchytné háky jsou proto nejdůležitější součástí systémů pro průběžnou integraci. Bez nich není možné automaticky reagovat na změny, které uživatelé vytvářejí a následně odesílají do sdíleného repozitáře.



Obrázek 2.11: Příklad záchytných háků

Háky na rozdíl od souborů nejsou sdílené, protože v každém repozitáři mohou být různé skripty spojené s různými událostmi. Nejčastěji je většina háků umístěna na serveru, který provádí testování, sestavovaní a podobné úkony.

2.1.5 Uchovávání historie změn

Uchovávání historie je další důležitou vlastností verzovacích systémů. Historie obsahuje hned několik položek:

- Jméno uživatele, který změnu provedl.
- Kdy změnu uživatel provedl.
- Popis změny, kterou uživatel provedl.

- Poslední změna, ze které uživatel vycházel.
- Obsah změny (co bylo odebráno a co přidáno).
- Vývojová větev, ve které se změna nachází.

Systém má jistá omezení, která chrání celistvost historie. Nelze například provést změnu bez popisující zprávy. Nelze vytvořit změnu, která má stejného předka jako jiná změna. Odesílána změna musí mít tedy v sobě zakomponované všechny změny, které již v repozitáři jsou. Jméno uživatele se může ověřovat a tím zjistit, zda uživatel má právo udělat změnu.

Ukázka možné podoby záznamu v historii (Mercurial):

```
changeset:   24:fd82dc9d5317
branch:      default
tag:         stable.2
user:        Vojta Tůma <ja@vojta-tuma.cz>
date:        Wed Dec 18 21:07:13 2013 +0100
summary:     Doc: added readme.md
```

Historie pak nalézá několik různých uplatnění:

- Dokážeme zjistit, kdo provedl jakou změnu.
- Dokážeme zjistit, kdo je autorem kódu, který upravujeme.
- Dokážeme zjistit, jaké změny nemáme zakomponované v lokálním repozitáři.
- Dokážeme se vracet ve změnách. Uvést kód do dřívější verze.
- Zobrazovat rozdíly mezi jednotlivými změnami.

- Kontrolovat produktivitu práce programátorů.

Shrnutím všech těchto vlastností dostáváme velice silný nástroj, jak kontrolovat vývoj projektu a provádět jeho verzování.

2.1.6 Dobré zvyky pro používání verzovacích systémů

Při používání verzovacích systémů existuje řada dobrých zvyků, které nám mohou velice ulehčit práci.

Asi nejdůležitější z nich je časté synchronizování kódu se sdíleným repozitářem. Pokud se nám během měsíce nahromadí změny od jiných uživatelů a my se budeme snažit tyto změny zakomponovat do naší lokální verze, je téměř jisté, že dojde ke konfliktu, který nám nedovolí změny automaticky zakomponovat. Musíme tedy ručně určit, jak se mají změny propojit. Toto nastane, pokud dva různí uživatelé editují stejnou část kódu.

Jedna změna by měla obsahovat pouze věci, které spolu souvisejí. Nedodržení tohoto pravidla vede k vytvářením dlouhých popisů změn, které se stanou nepřehledné. Většinou se však uchyluje k jednodušším zprávám jako *"velká změna"*, které jsou v podstatě nejhorší, protože o změně nic nevypovídají.

Výsledkem změny by měl být opět funkční kód. Pokud víme, že něco nefunguje naprosto správně, tak nemá smysl vytvářet změnu. Tím vznikají změny, do kterých se nelze vracet, protože v danou dobu byla aplikace nefunkční. Ne však vždy dokážeme určit funkčnost aplikace a provedeme nevědomě nefunkční změnu.

Provádíme-li opravu již implementovaných funkcí a zároveň přidáváme nové funkce, snažíme se nemíchat tyto dvě položky do jedné změny. Pokud ovšem na sobě nejsou funkčně závislé.

Posledním a nejdůležitějším zvykem jsou správně popisky změn. Popisek by měl obsahovat, jaké části aplikace se změna týká a co bylo opraveno nebo přidáno. Existuje dokument [3], který říká jak popisky vytvářet.

2.2 Představitelé verzovacích systémů

Existuje množství verzovacích systémů, kde každý vyniká v určitých oblastech. Pro práci byly vybrány systémy 3 nejpoužívanější [2] systémy a to GIT, Mercurial a Bazaar. Mezi další známější představitelé patří verzovacích systémů patří BitKeeper, CVS, Perforce a Team Foundation Server. V době vytváření práce existovalo celkem kolem 30 systémů pro verzování.

2.2.1 GIT

GIT je systém původně vyvinutý pro správu zdrojového kódu pro jádro Linuxu. Byl vytvořen Linusem Torvaldsem, který je zakladatelem Linuxu. Je založen na snímkování kódu. Pokud uživatel provede změnu na souboru, tak se vytvoří snímek souboru, který reprezentuje stav souboru před změnou. Ukládají se tedy vždy celé soubory. Hlavní vlastnosti GIT jsou :

- Vytváření snímků souborů.

- Do změn zařazovat pouze jednotlivé řádky.
- Možnost měnit historii.
- Není nutné stahovat celou historii.
- Nejvíce rozšíření (největší podpora).
- Mezi pracovní kopii a lokálním repozitářem je mezistupeň *stash*.
- Nevhodně zvolené názvosloví příkazů.

2.2.2 Mercurial

Mercurial, také označovaný jako HG, je výsledkem práce Matta Mackalla. Ten začal s vývojem ve stejnou dobu, jako se začal vyvíjet GIT. Většina systému je napsaná v jazyce Python, čímž se stává multiplatformní. Narozdíl od GIT používá pouze rozdílové změny, čímž je méně náročný na datový provoz. Hlavní vlastnosti Mercurial jsou:

- Intuitivní názvy příkazů.
- Jednoduchost.
- Změny pomocí *c*hangeset (seznam změn).
- Dobrá podpora.
- Vyspělé grafické nástroje.

2.2.3 Bazaar

Bazaar je systém podporovaný firmou Canoncial, která stojí za vývojem známe linuxové distribuce Ubuntu. Jedná se o multiplatformní systém napsaný v jazyce

Python. Vývoj se opět datuje do stejného období jako u dvou předchozích systémů. V poslední době systém neprojevuje větší vývoj a je spíš na ústupu. Nicméně z historických důvodů jsou pomocí něho verzovány projekty jako MySQL, MariaDB nebo GNOME.

2.3 Sestavovací systém

Ze zdrojového kódu je potřeba vytvořit spustitelnou aplikaci. Tento proces může být komplikovaný a ve výsledku se skládat z několik po sobě jdoucích činností, které je nutné manuál vykonávat. Ať už se jedná o kompilace, aktualizování databáze nebo kopírování souborů, jde téměř všechno zautomatizovat.

Vezměme si opět příklad, na kterém si vysvětlíme jakou má úlohu sestavovací systém s využitím predchozí znalosti o verzovacích systémech. Mějme webovou aplikaci, kterou upravujeme na lokálním počítači a musíme ji vystavit na veřejný web.

Úkol, který nás potká vždy, je kopírování nových souborů na úložiště, kde bude aplikace veřejně dostupná. Otevřeme tedy libovolného správce souborů a překopírujeme všechna potřebná data. Později zjistíme, že je lepším řešením je vytvoření skriptu, který úkony vykoná za nás. Ať už kopírujeme soubory skrze FTP, SSH nebo jiný protokol, jistě nalezneme i nástroj, který toto bude dělat za nás. Jedním ze základních nástrojů je synchronizace adresářů. Výsledkem je tedy skript, který spustíme ručně příkazem.

Dalším úkolem může být generování jazykových překladů pro aplikace. Pokud používáme gettext [5], je nutné soubor s překlady kompilovat. Při přidání nových překladů je nutné soubor ručně zkompilovat a následně ho nahrát společně s ostatními

soubory. Opět se jedná o věc, kterou můžeme přidat do skriptu. Přidejme ho tedy na začátek předchozího skriptu, aby se překlady zkompilovali a následně odešleme všechny soubory.

Pokud si chceme uchovávat změny v databázi, existuje možnost používat nástroj, který nám bude databázi automaticky aktualizovat. Nejjednodušší řešení je si ve svém projektu vytvořit složku, do které budeme přidávat soubory obsahující například SQL příkazy, které se mají nad databází vykonat. Lepší řešení je opět vytvořit skript, jenž nám tyto soubory sám spustí po nahrání. Přidejme ho tedy opět na začátek našeho prvního skriptu, protože je správné nejdřív upravit databázi a následně kód, který již počítá se změnami v databázi.

Aplikace je hojně používaná a nechceme, aby obsahovala chyby. Proto využíváme i testování, které se spouští pomocí několika nástrojů. Máme jednotkové testy, které v ideálním případě pokrývají většinu kódu naší aplikace. Tyto testy by měly být spuštěny před nahráním nového kódu na veřejný web. Opět je lze spouštět automaticky. Opět přidáme skript, který spustí všechny testy.

Právě jsme pokryli značnou část činností, které jsme museli provádět ručně, ale nyní je dělají skripty za nás. Tím jsme vytvořili automatický sestavovací systém. Jednotlivé části systému svážeme se záchytnými háky verzovacího systému. Nejprve spustíme testy a jako poslední nahrajeme aplikaci na veřejný web.

2.4 Testování

V předchozí kapitole bylo zdůrazněno testování softwaru. Jedná se o činnost, jejíž návratnost není tak zřejmá, protože testování přímo negeneruje novou funkčnost

aplikace, kterou by zákazník mohl vidět. Nicméně zkušenosti ukazují, že vytvářet testy má smysl[6].

Dobrým příkladem může být klasický internetový obchod. Zde je klíčovou vlastností možnost dokončení objednávky. Firma vytvoří pro zákazníka aplikaci, která bude prodávat jeho produkty na internetu. Již při vývoji několikrát zkouší, zda jde dokončit objednávka. Když vše funguje správně, tak produkt zákazníkovi prodají. Uplyne jistá doba a zákazník, chce vytvořit další funkčnost. Firma přidá nové funkce a opět ověří zda, objednávka funguje, protože se jednalo o velké zásahy. Vše se stále provádí ručně a každý test zabere neopomenutelnou dobu. Za rok se zákazník opět ozve a chce vytvořit zasílání vánočních poukázek. Termín na realizaci je velmi krátký, a proto firma narychlo nasazuje odesílání poukázek. Bohužel aktualizace vyžadovala úpravu komponenty, která odesílá e-maily, takže k odeslání e-mailu je nutný další povinný parametr. S tím ovšem nepočítá odesílání potvrzení objednávky. A výsledkem je, že se neodesílají e-maily po dokončení objednávky, což je klíčová vlastnost. Pokud by existoval test na odesílání e-mailu s potvrzením objednávky, nemohlo by k tomuto dojít. Při spuštění všech testů by došlo k chybě a sestavovací systém by nedovolil aktualizovat aplikaci.

Existuje několik způsobů testování [9]:

- Programátorské - Dva programátoři si navzájem kontrolují kód. Jeden kód vytvoří, druhý kód zkontroluje a vytvoří pro kód testy (zpravidla ručně otestuje funkce). Jedná se o nejméně nákladné a nejvíce účinné testování.
- Jednotkové - Následují po programátorském testování. Testovanou jednotkou lze rozumět jako oddělitelnou část aplikace, na kterou se vytvoří test, jenž ověří její funkčnost. Aplikovat jednotkové testy do existujícího projektu je většinou velmi složité. Pokud se tyto testy aplikují již při vývoji aplikace, vedou k vytváření kvalitní kódu.

- Funkční - Ověřují, zda je aplikace schopná plnit úkoly, pro které byla navržena a zda jsou splněny všechny požadavky zákazníka. V případě internetového obchodu by testování objednávky spadalo právě do funkčních testů.
- Integrační - Jedna z podmnožin funkčního testování. Kontroluje se správná komunikace komponent aplikace mezi sebou, ale i například komunikaci s operačním systémem nebo hardwarem. U menších projektů lze tuto část vynechat.
- Uživatelské - Aplikaci by měl pochopit i někdo jiný než její vývojář. Vezme se tedy náhodný člověk a zadají se mu úkoly, které má s aplikací vykonat. Pokud je nedokáže dokončit ani s návodem, je většinou aplikace špatně navržená.

Tím je ukončena motivační část, proč používat nástroje průběžné integrace.

3 Návrh vlastního řešení

Cílem práce bylo seznámení se s problematikou verzovacích systémů a nástroji průběžné integrace. Následně dané znalosti uplatnit pro návrh vlastního systému průběžné integrace, který je zaměřen na vývoj aplikací v prostředí www aplikací. Tento návrh měl být nakonec prakticky implementován.

3.1 Seznámení se s problematikou

Autor práce již před vytvořením diplomové práce měl zkušenost s používám verzovacích systémů. Nejvíce byl seznámen s verzovacím systémem Mercurial. Bylo nutné vylepšit znalosti o systému GIT a Bazaar.

GIT jako nejpoužívanější verzovací systém poskytuje širokou škálu dokumentačních materiálu. Existuje několik webových aplikací, které umožňují správu repozitáře skrze vzdálený přístup. Inspirovat se je možné například na projektu GitHub [7], který poskytuje jedno z nejlépe zpracovaných rozhraní pro správu repozitáře pomocí grafického rozhraní.

Stejně tak Mercuriál je velice dobře zdokumentovaný systém. Mezi nejznámější grafické nástavby patří HgTurtoise, který poskytuje všechny potřebné vlastnosti pro celkovou správu repozitáře. Autorova znalost Mercurialu však postačuje pro potřeby,

které bude výsledný systém obsahovat.

U systému Bazaar se ukázalo, že se již nadále aktivně nevyvíjí, a proto od jeho implementace do výsledného systému bylo upuštěno. Poslední aktualizace Bazaaru byla podle domovské stránky projektu [4] provedena v září roku 2013 a nadále se neplánuje jeho další vývoj.

Seznámení se s nástroji průběžné integrace nebo celými systémy bylo obtížnější. Existuje několik desítek systémů, které poskytují nástroje pro průběžnou integraci. Liší se v platformách, licencích, podpoře pro IDE a ostaních vlastnostech [8]. Mezi nejznámější systémy patří Jenkins, Travis a TeamCity. Všechny tyto systémy mají několik svých priorit, kterými se dá inspirovat. Podporují vystavení kódu na produkční server, spuštění testů, sestavení aplikace a mnoho dalšího.

3.2 Návrh vlastního řešení

Ze získaných znalostí a dosavadních zkušeností autora byl vytvořen návrh vlastního systému. Systém musí splňovat následující požadavky:

- Správa repozitářu
- Správa sestavovacích skriptů
- Správa přístupů
- Grafické rozhraní pro snadný přístup
- Konzolový přístup pro detailnější nastavení

Pro správu repozitářu je třeba vytvořit vlastní systém, který bude na vstupu podporovat dva různé druhy repozitářů. Repozitáře pro systém GIT a Mercuriál. Bude kladen důraz na jednoduchost nastavování repozitářů a možnost vykonávat základní operace nad repozitáři. Tedy vytvoření repozitáře, klonování repozitáře, zjištění potřebných údajů o repozitáři a případně i jeho smazání. Součástí systému pro správu repozitářů bude i možnost nastavování vlastních sestavovacích skriptů. Při vytváření skriptu bude kladen důraz na co největší intuitivnost při jeho vytváření. Bude použit návrh pomocí grafického editoru, který bude obsahovat funkční bloky. Tyto bloky bude uživatel spojovat a tak vytvářet složitější funkce.

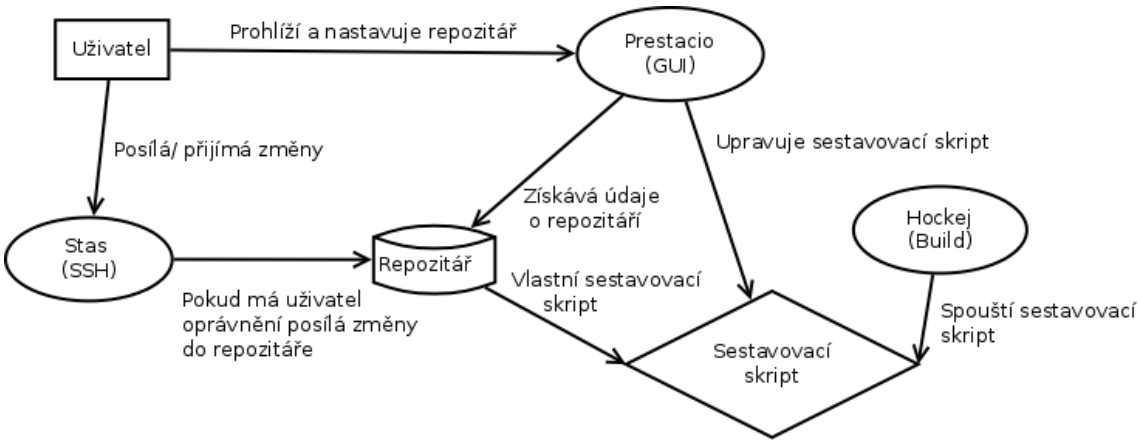
Pro správu přístupů k repozitářům bude vytvořen další systém, který bude spravovat přístupy pomocí protokolu SSH. Tento systém bude mít za úkol identifikovat příchozí spojení a ověřit, zda má povolení nad daným repozitářem provést požadovanou činnost. Jeho další činnost bude upravovat soubory operačního systému, které mají za úkol obsluhovat připojení pomocí protokolu SSH.

Pro možnost vytvoření sestavovacího skriptu bude nutné vytvořit vlastní jazyk, pomocí kterého se bude skript vytvářet. Dosavadní, autorem nalezené jazyky a nástoje, nejsou dostatečně ohebné, aby se daly jednoduše sestavovat v grafickém rozhraní. Zde se nachází asi nejsložitější část práce, které bude věnován největší důraz.

Jako hlavní programovací jazyk bude zvolen PHP, protože ho autor nejlépe ovládá. Sestavovací skripty budou ve formátu XML, protože tento formát poskytuje dobré nástroje pro validaci. Systém bude cílit hlavně na vývojáře používající volně šířitelný nebo svobodný software. Proto bude hlavním platformou operační systém Linux.

4 Vlastní řešení

V následující části práce se budeme zabývat implementací konkrétního systému, který si klade za cíl poskytnout uživateli všechny potřebné nástroje, aby mohl začít využívat průběžnou integraci pro svůj projekt. Zároveň klade důraz na jednoduchou instalaci, intuitivní ovládání a přehledné výstupy. Systém je rozdělen do tří částí. Jazyk je zvolena angličtina, aby nebyla aplikace omezená pouze pro národní trh. Skrze rozhraní, které aplikace poskytuje je možné provést lokalizaci do jakého koliv jiného jazyka.



Obrázek 4.1: Schéma systému

Obrázek 4.1 vystihuje schéma celého systému. Na počátku je uživatel, který má zájem poslat svoje nové změny do repozitáře. Připojí se tedy pomocí protokolu SSH ke vzdálenému repozitáři, kde Stas zkontroluje, zda má k připojení právo. Po ověření se dostává uživatel přímo k repozitáři, do kterého zapíše svoje

změny. V jiném případě se uživatel připojuje k Prestaciu, které mu umožní nastavení všech repozitářů. Získá informace o stavu repozitáře a má možnost měnit sestavovací skript přiřazený k danému repozitáři. Pokud dojde k události, která vyvolá spuštění sestavovacího skriptu, tak je spouštěn Hockej s příslušnými parametry. Všechny části Hockeje byly týmově vyvíjeny. Následuje popis činností jednotlivých komponent systému.

4.1 Hockej

Pro účel práce byl vyvinut sestavovací systém, který nese jméno Hockej. Lze pomocí něho spouštět různé cíle průběžné integrace. Počínaje kopírováním souborů a konče verzování databáze. Jeho hlavním cílem je co nejvíce zjednodušit tvorbu sestavovacích skriptů.

4.1.1 Sestavovací skript

Definice činnosti Hockeje je obsažena v sestavovacím skriptu. Tento skript obsahuje definice činností, které lze vykonávat. Soubor se skriptem má koncovku ".hockej", je ve formátu XML a jeho jednotlivé elementy mají speciální význam. Existují 4 druhý elementů:

- Definice - Nejmenší jednotka, která vykonává činnost (kopírování, mazání, filtrování atd.). Nelze samostatně spustit.
- Příkaz - Shluk více definic, které se sériově po sobě spouští. Jednotlivé příkazy je možné spouštět z příkazové řádky.
- Katalog - Obsahuje jednotlivé definice.

- Odvození - Z definice lze odvodit novou vlastní definici, která má již přednastavené vlastnosti.

Kořenovým elementem každého sestavovacího skriptu je element `<hockej>`, který říká, že se bude jednat o sestavovací skript Hockeje. Následuje nepovinný element `<description>`, který by měl popisovat činnost sestavovacího skriptu. Stejně tak může být doplněn nepovinný element `<version>` určující verzi daného skriptu.

4.1.2 Katalog

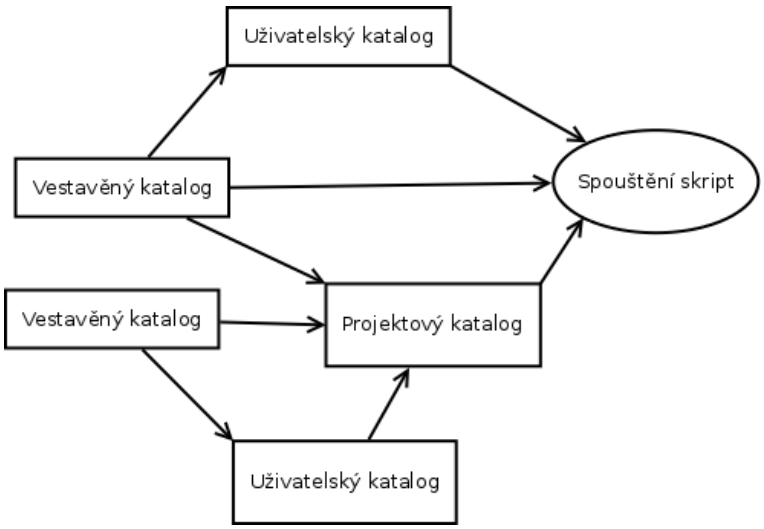
Definice všech činností, které Hockej dokáže vykonat, jsou umístění v katalozích. Katalog se do sestavovacího skriptu přidá pomocí elementu `<catalog>`. Má dva povinné parametry:

- from - Říká, kde se nacházejí soubory potřebné pro inicializaci katalogu.
- as - Přiřazuje katalogu jmenný prostor, pod kterým bude přístupný. Jeho elementy jsou následně přístupné jako "namespace."a jméno definice.

Existují celkem 4 zdroje katalogů:

- Vestavěné - Jsou přímo součástí instalace Hockeje. Kód, který se bude vykonávat je nadefinován pouze v nich. Ostatní katalogy pouze dědí z vestavěných katalogů. Výchozí adresář je "/usr/share/hockej/catalogs/".
- Uživatelské - Vlastní katalogy, které si vytvářejí uživatelé. Uložený zpravidla v uživatelské složce počítače. Výchozí adresář je "/home/user/.local/hockej/".
- Projektové - Jeden projekt může vlastnit několik privátních katalogů. Výchozí cesta není předem určená.

- Přímé - Zapisují se přímo do sestavovacího skriptu jako element `<catalog>` bez parametru "from".

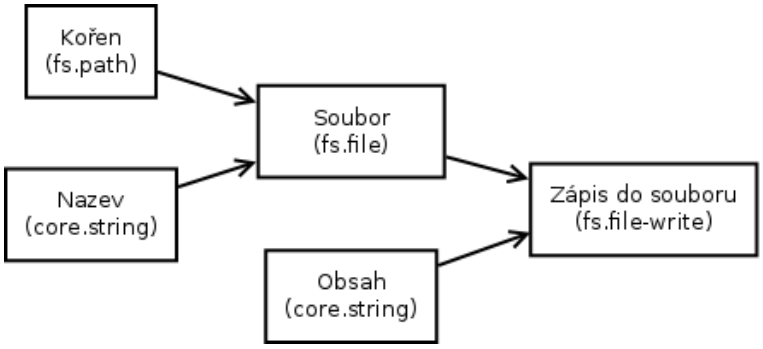


Obrázek 4.2: Hockej katalogy

Až na vestavěné katalogy, které jsou definovány více soubory, si všechny vytváří uživatel. Nejjednodušší je vytvořit přímý katalog, který je součástí sestavovacího skriptu.

4.1.3 Definice

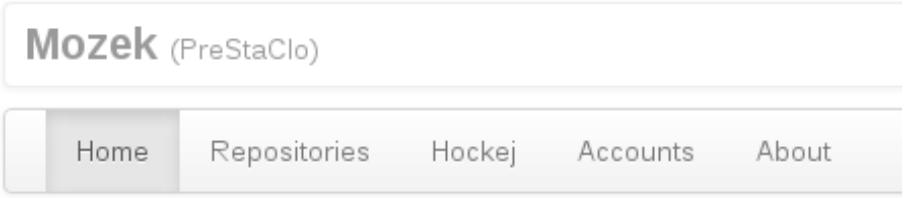
Nejmenší vykonatelnou částí pro Hockej je definice. Jedná se například o vytvoření instance souboru, zápis do souboru, vytvoření složky nebo spuštění skriptu. V předchozích případech byly definice například elementy `<print from="fs.file-write">`, `<file from="my.my` nebo `<content from="core.string">`. Všechny mají definovaný povinný parametr "from", který říká jaká je rodičovská definice toho prvku. Přičemž každý element může mít právě jednoho rodiče, ale i tak se dají vytvářet složité struktury.



Obrázek 4.3: Hockej - skládání definic

4.2 Prestacio

Aplikace poskytující grafické prostředí pro ovládání celého systému. Je vytvořena na základě aplikačního frameworku Nette [10]. Jedná se o nejdůležitější komponentu systému a proto je po ní celý systém nazván.



Obrázek 4.4: Menu

Prestacio se skládá z následujících částí:

- Home - Jednoduchý popis aplikace
- Repositories - Seznam všech repozitářů
- Hockej - Sestavovací systém (globální nastavení)
- Accounts - Uživatelské účty, pro přístup k repozitářům
- About - Místo pro vložení vlastního textu (například krátký popis firmy, projektů atd.)

4.2.1 Repozitáře

Prestacio dokáže spravovat několik repozitářů najednou. Skrze jednotlivé verzovací systémy jsou o repozitářích získáváné údaje, které jsou následně zobrazeny v grafickém prostředí uživately. Je možné spravovat repozitáře verzovacích systémů GIT a Mercuriál. Lze provádět pouze vytěžování informací a nelze provádět změny v nastavení repozitářů. Tím je částečně zabezpečena konzistence dat a zábráněno uživateli systému repozitář uvést do nefunkčního stavu.

4.2.2 Sestavovací skript

Prestacio umožňuje každému repozitáři vytvořit vlastní sestavovací skript, který slouží k zautomatizování čiností nad repozitářem. Skript může být vytvořen v textovém editoru nebo lze využít grafické nadstavby, která umožňuje jednoduché vytváření skriptů. Grafickému editoru a vytváření skriptů bude věnována část další kapitoly.

4.2.3 Hockej

Tato část získává všechny dostupné informace o aktuálním nastavení Hockeje. Tedy informace o katalozích, kde jsou podrobně popsány všechny definice, které jsou v katalozích obsaženy. Katalogy jsou rozděleny na systémové a uživatelské.

4.2.4 Uživatelské účty

K repozitářům mohou přistupovat rozličtí uživatelé, proto tato sekce umožňuje spravovat přístupy uživatelů k jednotlivým repozitářům a přidávat k nim oprávnění na čtení či zápis. Je zde také správa všech veřejný klíčů potřebných pro zabezpečenou komunikaci protokolem SSH mezi uživatelem a repozitářem.

4.3 Stas

Pro možnost připojení uživatelů skrze protokol SSH bylo nutné vytvořit systém, který bude uživatele ověřovat. Jedná se tedy o vrstvu, která je vložena mezi službou spravující SSH komunikaci a repozitářem. Zkratka systému Stas znamená "Ssh Trivial Authorization System". Jedná se o obdobu systému Gitolite [11], který ovšem podporuje pouze verzovací systém GIT.

V případě, kdy se uživatel připojuje skrze prokol SSH ke vzdálenému repozitáři a připojení není spravované pomocí Stas, je průběh následujicí:

- Uživatel se autorizuje pomocí svého veřejného klíče nebo hesla službě serveru na který se připojuje.
- Uživatel dostane přístup přímo k repozitáři a nahraje do něj svoje změny.

Není tedy zde nic, co by uživatele identifikovalo. Do repozitáře tedy mohou přispívat všichni, kteří mají heslo nebo zaregistrovaný veřejný klíč.

V druhém případě se uživatel připojuje ke vzdálenému uložišti spravovaného pomocí Stas, které zajistí indentifikaci uživatele a zkontroluje, zda má přístup pro danou operaci na repozitářem. Postup je tedy následujicí:

- Uživatel se autorizuje pomocí svého veřejného klíče serveru na který se připojuje.
- Spustí se Stas, které podle klíče zkontroluje uživatele připojujícího se k repozitáři.
- Stas zjistí, zda má daný uživatel právo k repozitáři přistoupit.

- Stas zkontroluje o jakou činnost se bude jednat. Existují 2 typy (čtení a zápis).
- Stas povolí nebo zamítne činost.

Aby bylo možné připojovat uživatele pomocí SSH protokolu, musí Stas nakopírovat veřejné klíče všech uživatelů zadaných v Prestaciu do souboru `~/.ssh/authorized_keys`.

5 Návod na používání a praktické ukázky

5.1 Hockej

V této kapitole bude popsáno jakým způsobem se používá navržený systém pro vytváření a vykonávání sestavovacích skriptů.

5.1.1 Instalace

Jedná se o samostatnou součást, která vyžaduje instalaci. Pro instalaci je nutné mít nainstalovaný Composer(`https://getcomposer.org/`), GIT(`http://git-scm.com/`) a PHP ve verzi větší než 5.4. Následně je třeba naklonovat repozitář s Hockejem z přiloženého CD nebo z aktuálního repozitáře pomocí příkazu `git@bitbucket.org:Tacoberu/hockej.git`, přepnout se do složky s naklonovaným hockejem a spustit Composer `composer install`. Hockej musí být umístěň do adresáře, kde bude globálně spustitelný.
Například `ln -s source/bin/hockej ~/bin/hockej`.

5.1.2 Vytvoření sestavovacího skriptu

Definice katalogů, které bude skript využívat se definují pomocí elementu `<catalog>` s povinnými parametry `from`(zdroj katalogu) a `as`(jmenný prostor).

Nejjednodušší katalog může tedy vypadat následovně:

```
<?xml version="1.0" ?>
<hockej>

  <description>First buildscript</description>

  <version>1.2.3</version>

  <catalog from="hockej://build-in/hockej/filesystem" as="fs" />

</hockej>
```

Skript si při spuštění inicializuje katalog "filesystem"a přiřadí mu jmenný prostor "fs". Nic víc však neudělá. Je proto nutné definovat příkaz pomocí elementu `<command>`, který bude moci Hockej spustit. Element musí mít nadefinovaný atribut `name`, který říká jakým příkazem bude spuštěn. Nadefinujme si tedy příkaz, který zapíše do souboru "priklad.txt"řetězec "Hello, I'am Hockej!". Výsledný skript bude vypadat následovně:

```
<?xml version="1.0" ?>
<hockej>
  <description>First buildscript</description>
  <version>1.2.3</version>

  <catalog from="hockej://build-in/hockej/filesystem" as="fs" />
  <catalog from="hockej://build-in/hockej/core" as="core" />

  <command name="build">
```

```
    <print from="fs.file-write">
      <file from="fs.file">
        <root from="fs.path">.</root>
        <filename from="fs.file">priklad.txt</filename>
      </file>
      <content from="core.string">Hello, I'am Hockej!</content>
    </print>
  </command>
</hockej>
```

Po spuštění sestavovacího souboru příkazem "hockej build"ve složce se skriptem Hockej zapíše text do souboru. Nyní si podrobně probereme, co jednotlivé položky znamenají.

5.1.3 Vytvoření vlastního katalogu

Pro příklad si, ale uvedeme definici katalogu v externím souboru, která bude prakticky stejná.

Externí katalogy jsou definované v souborech s názvem "catalog.hockej"a jsou ve formátu XML. Vytvořme jednoduchý katalog, který nám bude definovat cestu k souboru "priklad.txt":

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>

  <id>routes</id>
```

```
  <version>0.0.4</version>

  <description>Cesta k˜souborům</description>

  <catalog from="hockej://build-in/hockej/filesystem" as="fs" />
  <catalog from="hockej://build-in/hockej/core" as="core" />

  <catalog>
    <my-file from="fs.file">
      <root from="fs.path">.</root>
      <filename from="core.string">priklad.txt</filename>
    </my-file>
  </catalog>

</catalog>
```

Pokud si nyní přidáme do sestavovací skriptu tento katalog, budeme mít k dispozici nový element `<my-file>`.
Pomocí `<catalog from="hockej://project//routes" as="my" />` přidáme vlastní katalog do skriptu a můžeme používat následující předpis pro zápis do souboru :

```
  <command name="build">
    <print from="fs.file-write">
      <file from="my.my-file">
      <content from="core.string">Hello, I'am Hockej!</content>
    </print>
  </command>
```

Soubor s katalogem by v tomto případě byl uložen ve složce project/routes, která by se nacházela v kořenovém adresáři projektu. Tím se dá ulehčit opisování stejných konstrukcí pro nazvání souboru, vytváření složek a ostatních.

5.1.4 Ukázka rozšiřování definic

Například definice "fs.file-write"zajišťuje zápis do souboru. Vytvořili jsme tedy element s libovolným názvem, který dědí od definice "fs.file-write". Příkladem může být element `<print from="fs-file-write">` z minulého příkladu. Tato definice vyžaduje další dvě definice:

- Soubor, do kterého se bude zapisovat. Jedná se o potomka definice "fs.file".
- Text, který se bude zapisovat. Zde se bude jednat o potomka definice "core.string".

Tyto závislosti je možné získat pomocí parseru katalogů, ten je v grafické podobně implementován v Prestaciu. Závislosti umožňují vytvářet složité struktury, které se mohou různě překrývat. Tato vlastnost je klíčová, kvůli které byl Hockej vytvořen. Lze předdefinovávat již nastavené definice nebo je doplňovat. Nejlépe si to můžeme vysvětlit na příkladu se souborem, tedy s potomkem definice "fs.file", který vyžaduje následující definice:

- Kořen - Potomek definice "fs.path"
- Cesta - Potomek třídy "fs.path"
- Název souboru - Potomek třídy "core.string"

Jeho definice může vypadat následovně :

```
<soubor from="fs.file">
  <path from="fs.path">/korenovy/adresar/</path>
  <path from="fs.path">/cesta/dal/</path>
  <filename from="core.string">nazev_souboru.txt</filename>
</soubor>
```

Tím jsme vytvořili definici, kterou můžeme použít k vytvoření elementu `<muj-soubor from="s` Potřebujeme vytvořit další definici, ale chceme pouze změnit jméno souboru. To lze uskutečnit následovně:

```
<druhy-soubor from="soubor">
  <filename from="core.string">jiny_nazev.txt</filename>
</druhy-soubor>
```

Tuto definici může opět použít pomocí elementu `<jiny-sobour from="druhy-soubor">` a předat definici pro zápis do souboru a následně libovolně měnit její předky.

5.1.5 Spuštění

Samotný Hockej se při správně provedené instalaci spouští příkazem `hockej` a parametrem, který říká co má udělat:

- `help` - Zobrazí plnou nápovědu.
- `list` - Zobrazí seznam všech spustitelných příkazů. Atributy "name"od všech elementů `<command>`.

- `run <příkaz>` - Vyhledá v aktuální složce soubor "build.hockej"a spustí v něm příkaz. Například `hockej run build`.
- `-f --buildfile <nazev_skriptu>` - Specifikuje název souboru se sestavovacím skriptem. Například `hockej run build -f build-local.hockej`.
- `--xml` - Výstup ve formátu XML. Například `hockej run build --xml`.
- `--dry-run` - Provede suchý běh. Fyzicky se nic nevykoná.
  Například `hockej run build --dry-run`.

5.2 Prestacio

5.2.1 Instalace

Prestacio je webová aplikace, která pro svoji instalaci vyžaduje webový server, který obsahuje interpreter jazyk PHP pro verzi 5.4 a vyšší. Dále je nutný nainstalovaný Composer a GIT. Následně je třeba naklonovat repozitář s projektem z přilozeného CD nebo z aktuálního repozitáře pomocí příkazu
`git clone git@bitbucket.org:Tacoberu/prestacio.git`. Přepnout se do složky s naklonovaný projektem a spustit Composer příkazem `composer install`, tím se doinstalují všechny závislosti. Jako poslední krok je nutné vytvořit instanci hosta v konfiguraci webového serveru, která bude odkazovat do složky webroot.

Součástí instalace Prestacia je i instalace Stas.

5.2.2 Repozitáře

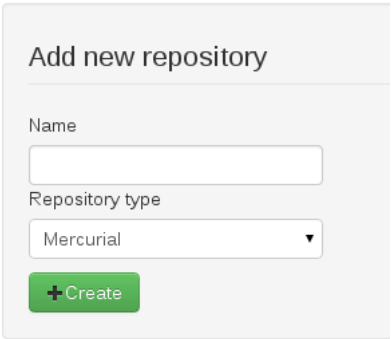
Prestacio dokáže hospodařit s několika repozitáři nezávisle na sobě. U každého repozitáře je zobrazen jeho název a příkaz, pomocí kterého je možné si vytvořit lokální kopii repozitáře. Poslední položka je stav repozitáře, který znázorňuje stav posledního

sestavení aplikace. Pokud se sestavení zdařilo je zobrazena 1 v opačném případě je zde nula.

Obrázek 5.1: Seznam repozitářů

Zatím podporované verzovací systémy jsou GIT a Mercurial. Podle zkratek *git* a *hg* lze rozeznat, kterým verzovacím systémem je repozitář spravován.

Pomocí tlačítka *Add new repository* je možné přidat nový repozitář ke správě.



Obrázek 5.2: Přidat repozitář

V přidávacím formuláři stačí zvolit název repozitáře a verzovací systém, pomocí kterého bude spravován. Všechny tlačítka v Prestaciu jsou barevně odlišené:

- Zelená - Vytvoření nové prvku.
- Modrá - Editace existujícího prvku.

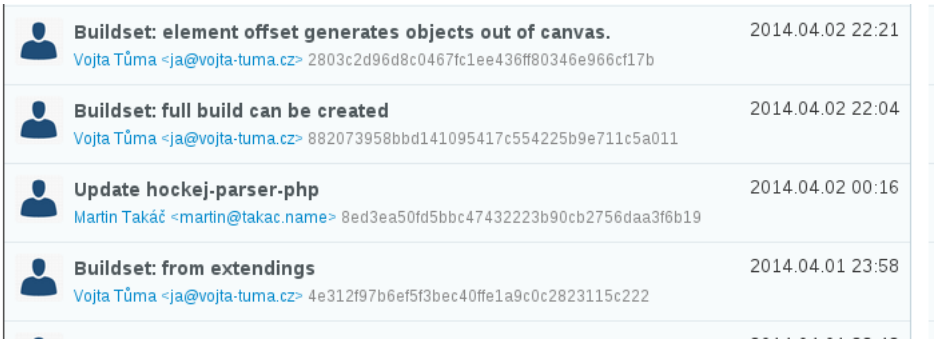
- Šedá - Zobrazení podrobností.
- Červená - Smazání prvku.

Po kliknutí na název repozitáře se otevře detailní zobrazení repozitáře.

Obrázek 5.3: Detail repozitáře

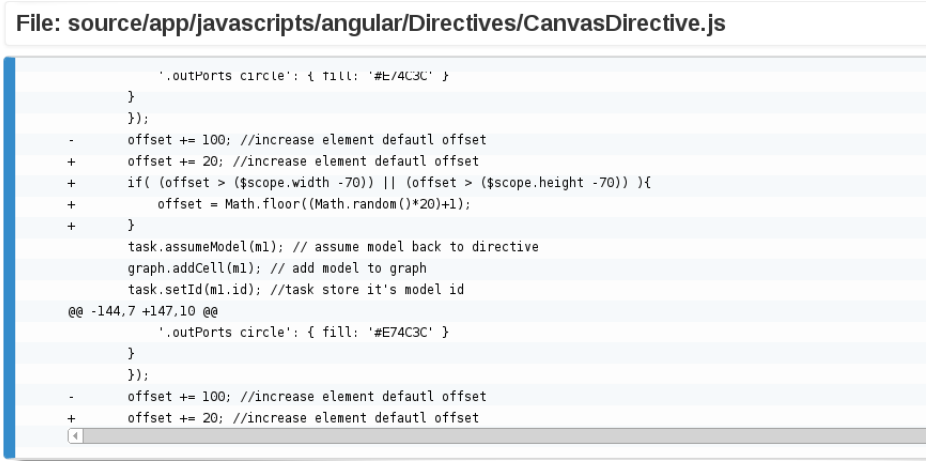
Kde jsou vypsány jednotlivé údaje o repozitáři:

- Name - Jméno repozitáře.
- Url - Adresa, pomocí které je možné repozitář naklonovat.
- Last tag - Poslední přidělená značka.
- Last commit - Záznam o poslední nahrané změně. Obsahuje jméno uživatele, kontrolní součet v případě GIT, popisek změny a možnost zobrazit diferenciální zobrazení s předchozí změnou.
- ACL - Omezení přístupových prav.
- Deploy status - Stejný význam jako status v seznamu repozitářů a možnost zobrazit detailní zprávy s výsledky sestavovacího systému.
- Buildscript - Odkaz na definování vlastního sestavovacího skriptu.



Obrázek 5.4: Změny v repozitáři

Pro lepší přehled vývoje kódu je důležité vidět seznam všech změn, které byly v repozitáři provedeny. Prestacio obsahuje přehledné zobrazení, které poskytuje základní informace o všech změnách, které byly v kódu provedeny. Všechny tyto informace poskytuje verzovací systém a Prestacio je převádí do přehlednější podoby. Na obrázku 3.5 je vidět, že Vojta Tůma provedl změnu s popisem: "Buildset: element offset generates objects out of canvas.". Tato změna byla provedena 2.dubna v 22:21. Po kliknutí na popis změny se zobrazí detailnější informace o změně.



Obrázek 5.5: Detail změny

V detailním popisu jsou zobrazeny jednotlivé soubory, jejichž obsah byl v dané změně upraven. Konkrétní zobrazení je závislé na použitém verzovacím systému, protože výstup je generován přímo verzovacím systémem. Na obrázku 3.6 vidíme,

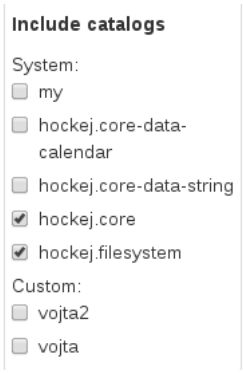
že byl změněn obsah souboru CanvasDirective.js. Řádky, které začínají znamínkem mínus říkají, že byl daný řádek v souboru odebrán. Naopak řádky začínající znaménkem plus značí přidání nového řádku v rámci změny.

5.2.3 Sestavovací skript

Každému repozitáři je přiřazen jeden sestavovací skript. Po kliknutí na odkaz "Edit buildscript"v detailu repozitáře se zobrazí okno s možností editace sestavovací skriptu Hockeje.

Celé prostředí je rozdělené do 3 panelů:

- Levý - Definuje se v něm popisek, verze a katalogy, které bude sestavovací skript používat.
- Střední - Vkládají se do něj elementy, ze kterých se sestavuje výsledný skript. Zobrazuje se v něm výsledné xml.
- Pravý - Zobrazují se v něm možné parametry elementů.

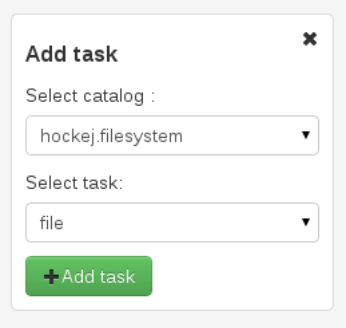


Obrázek 5.6: Přidávání katalogů

Nejdříve vybereme v levém panelu katalogy, které chceme používat. Jsou rozdělené na "System"(vestavěné) a "Custom"(projektové a uživatelské). Nad středním panelem je lišta, na které jsou 4 tlačítka:

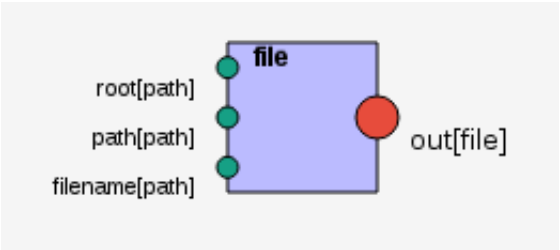
- Add task - Přidá novou definici.
- Add command - Přidá nový příkaz.
- Graphic editor - Zobrazí sestavovací skript v grafické podobě.
- Xml preview - Z grafické podoby vygeneruje výsledný XML soubor.

Novou definici přidáme kliknutím na tlačítko "Add task". Zobrazí se dialogové okno, ve kterém zvolíme z jakého katalogu chceme definici vybírat, následně vybereme danou definici a klikneme na tlačítko pro přidání definice do grafického editoru.



Obrázek 5.7: Přidávání definice

Po vložení se zobrazí element v grafickém editoru. Na levé straně má definice, které vyžaduje, aby mohl správně pracovat. Na pravé straně jaký jeho výstup. Na levé straně vidíme, že vyžaduje parametr "root", který je typu "path". Podle názvu zjistíme, že se bude zřejmě jedna o kořenový adresář. Další dva parametry jsou opět typu "path". Jeden je relativní cesta od kořene dále a druhý je název souboru. Výstup na pravé straně ukazuje, že z toho elementu je možné získat definici typu



Obrázek 5.8: Definice file

”file”.

Nyní existují dva způsoby, jak dané definici file dodat jí požadované parametry. První způsob je kliknutí na daný element, kde se následně v pravém sloupci zobrazí definice, které lze nastavit ručně. Většinou to jsou definice, které se dají reprezentovat jako textové řetězce. V tomto příkladu to jsou všechny 3 definice. A v pravém panelu máme možnost nastavit všechny 3.

Nastavili jsme kořenový adresář ”root”a název souboru ”filename”. Nyní je definice kompletní a při spuštění jí Hockej dokáže zpracovat. Tím, že jsme určili následující definice ručně se změnila podoba grafického elementu v editoru. Za definice, které jsou vyplněny ručně se vepsal znak ”@”.

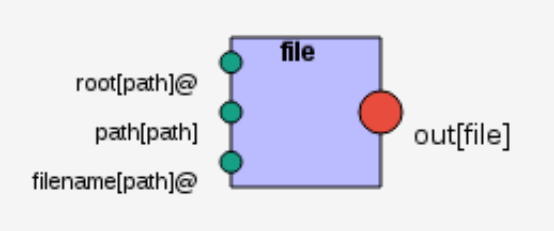
Tímto způsobem však nelze nastavit všechny definice. Například definice pro čtení ze souboru požaduje definici typu ”file”, která se nedá zapsat jako textový řetězec. Pro tento důvod slouží propojování jednotlivých definicí v grafickém editoru. Lze propojit výstup jedné definice se vstupem druhé definice, pokud souhlasí jejich typy ve hranatých závorkách. Ukažme si tedy, jak se například dá vytvořit definice typu ”file”pomocí více definice. Pomocí tlačítka ”Add task”přidáme definici pro čtení ze souboru, která se nachází v katalogu ”hockej.filesystem”. Stejným způsobem si přidáme i definici souboru ”file”, která se nachází opět ve stejném katalogu. Definici souboru nastavíme kořenový adresář a název souboru jako v minulém

Obrázek 5.9: Nastavení definice

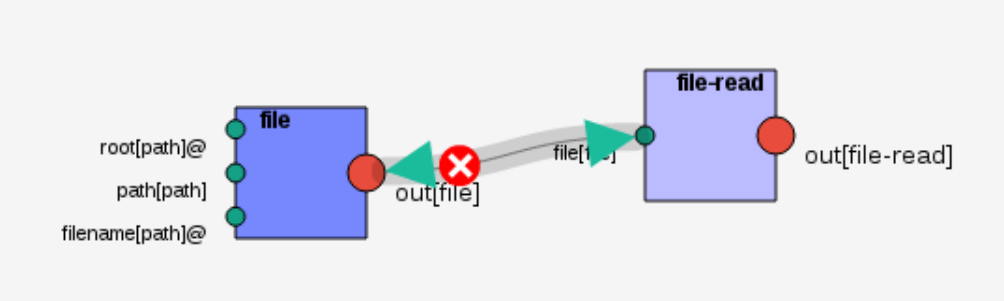
případu a její výstup propojíme se vstupem definice pro čtení souboru.

Propojení provedeme kliknutím myší na výstup a přetažením na vstup jiné definice. Dané propojení lze zrušit po kliknutí na červené kolečko s křížkem. Tím jsme vytvořili definici, která dokáže číst ze souboru. Tato definice může být následně předána jiné definici a tak dále. Tímto způsobem je možné vytvářet složitější struktury.

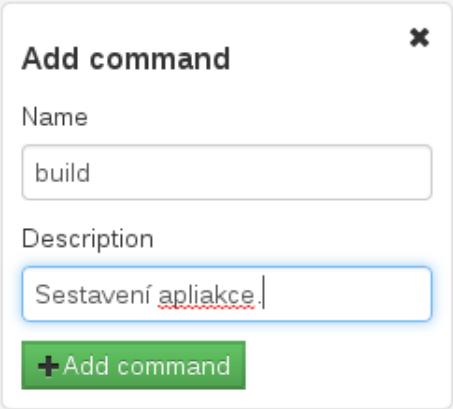
Aby mohly být definice spuštěné, je nutné vytvořit příkaz, který bude pomocí Hockej spustitelný. Na toto slouží tlačítko "Add command", které přidá do grafického editoru elementu tu příkaz. Od definice se liší barvou, neomezeným počtem vstupů a žádným výstupem.



Obrázek 5.10: Ručně upravená definice

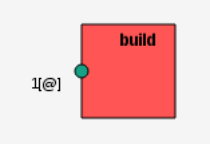


Obrázek 5.11: Propojení definic



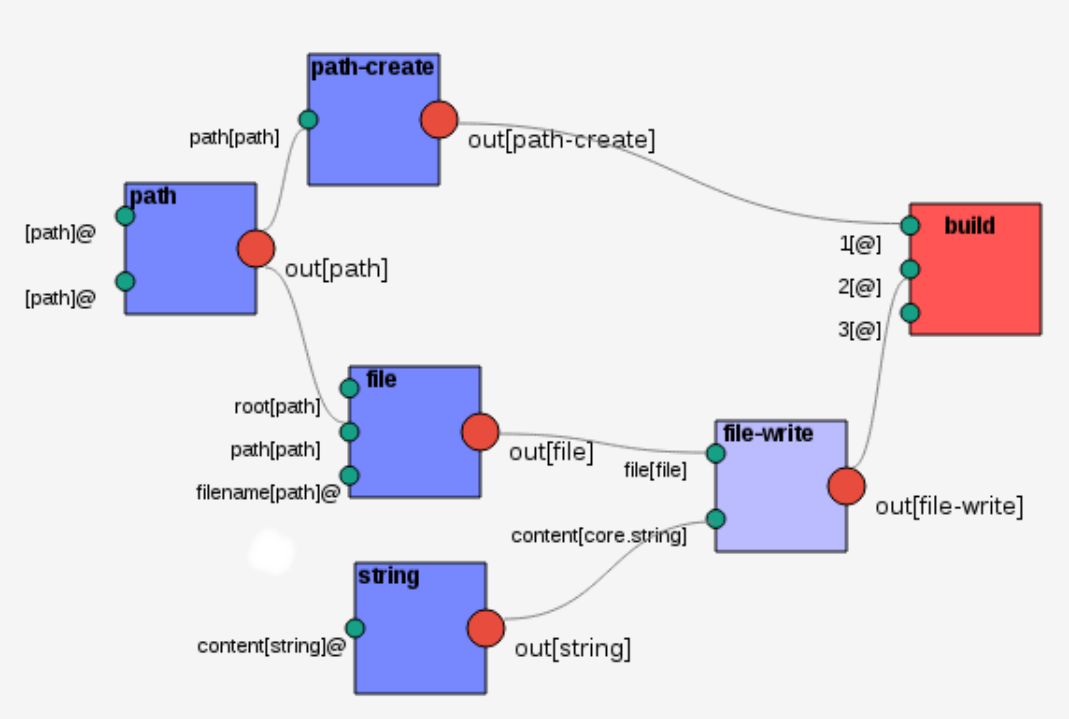
Obrázek 5.12: Dialog vytvoření příkazu

Vytvoření příkazu vyžaduje zadání jména příkazu, pod kterým bude pro hockej přístupný, popisek, který bude danou akci popisovat a záchytný hák, který ho bude spouštět. Zvolme tedy například název akce "build"a popisek pro ní bude "Sestavení aplikace". Příkaz vložíme kliknutím na tlačítko "Add Command".



Obrázek 5.13: Příkaz

Na vstupu příkazu se zobrazí 1[@]. Číslo na vstupu znamená pořadí spouštěné definice a znak "@"v hranatých závorkách, že do jeho vstupu lze připojit jakoukoliv definici. Vytvořme příkaz, který vytvoří složku a do ní vytvoří soubor.



Obrázek 5.14: Ukázka sestavení

Skript začne úplně vlevo, kdy se vytvoří definice "path", která poskytne cestu pro definici "path-create"sloužící pro vytvoření adresáře. Ta je následně připojena k příkazu "build"na první pozici. Tato definice bude tedy spuštěna jako první. "Path"také poskytuje cestu pro definici "file"a ta je následně spojena s "file-write", která zapisuje text do souboru. "File-write"ovšem potřebuje text, který se bude do souboru zapisovat. Tento text je dodán pomocí definice "string"obsahující řětězec,

který se bude zapisovat. "File-write"je následně propojena s příkaz "build"na druhé pozici. Provede se tedy jako druhá. Ve výsledku tedy vytvoříme adresář, do kterého následně zapíšeme soubor s předdefinovaným textem.

Z grafického editoru je nyní možné vytvořit sestavovací skript pro Hockej. Stačí kliknout na tlačítko "Xml Preview"v horní liště. Následně se vygeneruje soubor, který je možné pomocí Hockeje spustit.

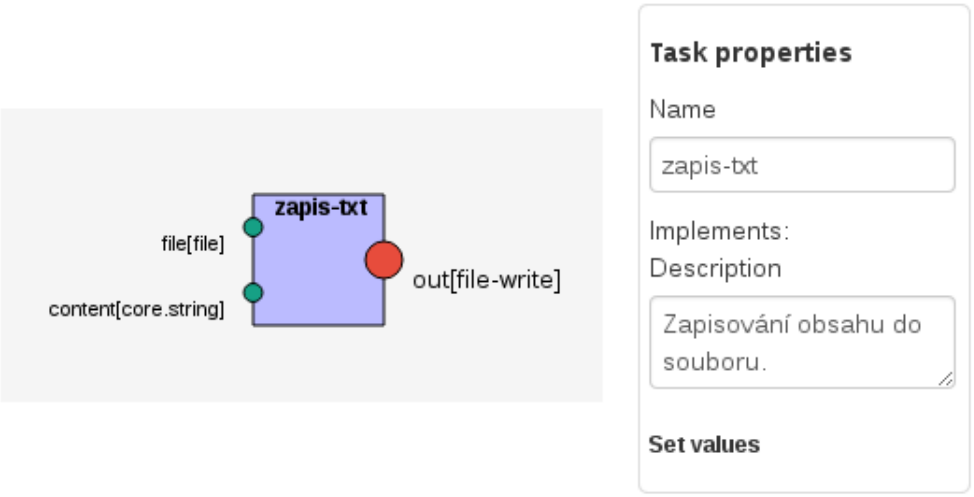
```
<hockej>
<description>Ukazka</description>
<version>0.0.1</version>
<catalog src="hockej://build-in/hockej/filesystem" as="fs" />
<catalog src="hockej://build-in/hockej/core" as="core" />
<catalog>

  <path description="Element pro odkazování cesty." from="fs.path">
    <root from="fs.path">/home/prestacio/</root>
  </path>

  <path-create description="Vytvoření adresáře." from="fs.path-create">
    <path from="path" />
  </path-create>

  <file-write description="Zapisování obsahu do souboru." from="fs.file-write">
    <file from="file" />
    <content from="string" />
  </file-write>

  <file description="Reprezentace jednoho souboru." from="fs.file">
    <path from="path" />
```

```
      <filename from="fs.path">pokus.txt</filename>
    </file>

    <string from="string">
      <content from="core.string">Hello, I'm Hockej!</content>
    </string>

  </catalog>

  <command name="build" description="Sestavení aplikace">
    <path-create from="path-create"/>
    <file-write from="file-write"/>
  </command>

</hockej>
```

Nyní jsme vytvořili kompletní spouštěcí skript, který stačí pomocí tlačítka "Save script"uložit. Pokud bychom chtěli ale vytvořit skript, který bude zapisovat do dvou různých souborů, tak nastane problém. Definice "file-write"už jednou existuje. Vložíme do editoru další definici "file-write", která se bude jmenovat stejně a připojíme ji do příkazu, Hockej nebude vědět, kterou definici jsme měli na mysli. Logika Hockeje je taková, že spustí poslední definici, na kterou narazí. Proto Prestacio umožňuje vytvářet vlastní názvy definic. Stačí kliknout na element v grafické editoru a do políčka "name" napsal vlastní název definice. Definice se přejmenuje, ale její předek zůstane stejný.

Definice se do výsledného kódu pro Hockej zobrazí jako:

```
  <zapis-txt description="Zapisování obsahu do souboru." from="fs.file-write">
```



Obrázek 5.15: Ukázka sestavení

```
    ...
</zapis-txt>
```

Tuto definici je dále možné používat v jiných definicích, kde k ní přistoupíme jako `<file-write from="zapis-txt" />` nebo `<muj-zapis from="zapis-txt">`. Hockej podle předka pozná o jakou definici se jedná.

Hockej poskytuje základní definice jako vytvoření souboru až po nahrávání souborů skrze SSH protokol na vzdálené úložiště. Obsahuje také definice pro spouštění PHPUnit testování. Autor práce počítá s budoucím rozšířením Hockeje o další definice a o možnost vytváření čistě uživatelských definic.

5.2.4 Výsledek sestavovacího skriptu

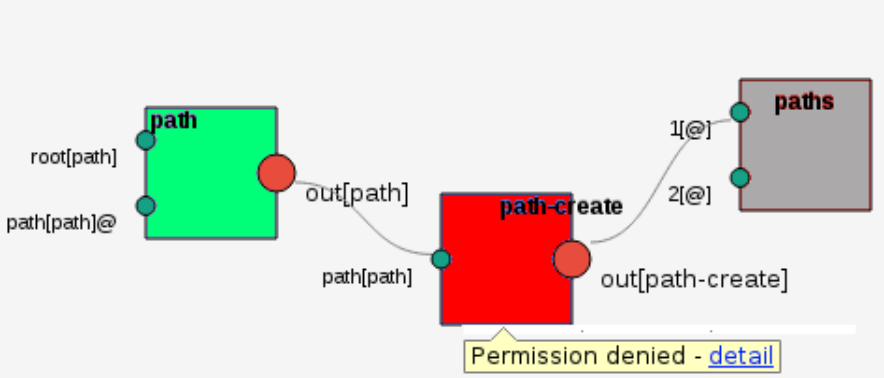
Po vykonání sestavovacího skriptu je důležité zjistit, zda se všechny operace dokončily správně. Z toho důvodu je Prestacio vybaveno zobrazením, které podrobně

popisuje stav vykonání všech definicí. Přistoupit k výsledkům skriptů je možné v detailu repozitáře skrze odkaz "logs". Číslo před odkazem značí, zda bylo poslední sestavení úspěšné nebo ne.

Po otevření stránky se nám zobrazí okno, které poskytuje dva možné výstupy. Pomocí tlačítek "Raw" a "Graphical" můžeme přepínat mezi čistým textovým výstupem, který je ve formátu XML nebo grafický výstupem totožným s grafickou podobou vytvořenou v editoru sestavovacího skriptu. Tentokrát jsou grafické elementy jinak zbarveny a jsou doplněny o informace, zda byly správně vykonané. Barvy značí následujicí stavy:

- Zelená - Definice se v prořádku vykonala.
- Červená - Definice se nevykonala, protože došlo k chybě.
- Šedá - daný cíl se nevykonal, protože jedna z jeho definic neuspěla.

Výsledkem nejzákladnějšího skriptu může být například zobrazení na obrázku 5.16. Skript dokázal vytvořit definici pro cestu, protože obsahovala povinný parametr "path". Dále se snažil skript vytvořit cestu pomocí definice "path-create". Tato definice již bohoužel neuspěla a důvod neúspěchu oznámila hláškou "Permission denied". Zřejmě Hockej nemá právo na vytvoření dané složky. Pod odkazem detail je možné nálezt detailní popis celé akce, kterou se snažila definice vykonat. Na konec je příkaz "paths" vybarven šedě, protože jedna z jeho definic nebyla vykonána a příkaz tam nebyl kompletně dokončen.



Obrázek 5.16: Výsledek sestavení

5.2.5 Katalogy Hockeje

Aby bylo možné zjistit, které katalogy poskytují jaké definice, je součástí Prestacia i modul, který dokáže vyčíst informace ze všech katalogů zaregistrovaných v Hockeji. Tento modul je přístupný skrze hlavní menu pod položkou Hockej.

Následně se zobrazí všechny vestavěné a uživatelské katalogy. U každého katalogu je jeho název, popis a odkaz na podrobnosti.

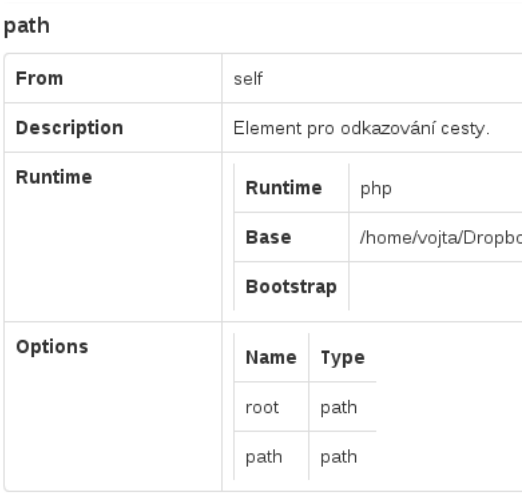
- hockej.core-data-string - Práce s řetězci.[detail]
- hockej.core - Základní knihovna typů - string, numeric, real, boolean, empty, any.[detail]
- hockej.filesystem - Základní knihovna typů, filterů a tasků pro práci se soubory.[detail]

Obrázek 5.17: Seznam katalogů

V detailu každého katalogu lze zjistit jaké definice katalog obsahuje, co definice vykonávají a jaké jsou jejich závislosti.

Jak je vidět na obrázku 5.18 každá definice obsahuje:

- From - Kdo je předkem definice.



Obrázek 5.18: Detail katalogu

- Description - Krátký popis definice.
- Runtime - V jakém běhovém prostředí bude definice spuštěna.
- Option - Parametry definice, které ji lze nastavit.

Pomocí tohoto zobrazení je možné definice ručně vytvářet bez použití grafického editoru. Lze tedy vytvářet přímo soubory katalogů pro hockej skrze textový editor, což je zatím jediná cesta, jak vytvářet uživatelské a projektové katalogy.

5.2.6 Uživatelské účty

Pod položkou "Accounts"se skrývá správa uživatelských účtů. Jsou zde zaregistrování všichni, kteří přistupují pomocí protokolu SSH k repozitářům. V přehledu je u každého uživatele vždy uveden e-mail, jméno a přezdívka.

Po kliknutí na jméno uživatele se zobrazí detail uživatele.

Obrázek 5.19: Seznam uživatelů

Obrázek 5.20: Detail uživatele

Zde je přehled daného uživatele, který obsahuje:

- Položky v přehledu (jméno, přezdívku, a všechny registrované e-mailové adresy)
- Repozitáře ke kterým má přístup a práva k nim (zápis, čtení)
- Veřejné klíče pro komunikace skrze SSH

Ke každému uživateli je možné přidat neomezené množství veřejných klíčů, pomocí kterých se následně připojuje ke vzdálenému repozitáři.

6 Závěr

Při vytváření systému se autor snažil využít všechny svoje dosavadní zkušenosti, které získal během práce na velkých webových projektech. Hlavním cílem bylo vytvořit systém, který budou moci používat i méně zkušení uživatelé, kteří se v oblasti průběžné integrace příliš neorientují.

6.1 Složitost vytvoření systému

Systém je rozdělen do 3 částí, které jsou na sobě prakticky nezávislé a mohou se dále samostatně vyvíjet. Všechny části poskytují velmi dobrý základ, který se dále může rozšiřovat. Část zodpovědná za SSH komunikace je nejméně složitá. Koncept byl již vytvoření a stačilo dodělat podporu pro více verzovacích systémů. Prestacio jakožto grafické rozhranní aplikace obsahuje několik rozličných částí, které zahrnují vytěžování informací z verzovacích systémů, správu uživatelů a vytváření sestavovacích skriptů. Nejsložitější z uvedených je vytváření sestavovacích skriptů, které ovšem poskytuje velmi jednoduchý nástroj pro zautomatizování cílů průběžné integrace. Sestavovací skripty pak vykonává Hockej. Část, která za dobu vytváření práce byla několikrát od základu přepsána, aby byla co nejvíce pochopitelná a uživatelsky přívětivá.

6.2 Intuitivnost ovládání

Velký důraz při vytváření systému byl kladen na intuitivnost ovládání. Všechny položky systému byly tedy pokud možno převedeny do grafické podoby, která se zdá být přehlednější než textový výstup. Většina existujících systémů pro průběžnou integraci má jako vstup dlouhé nepřehledné textové soubory a stejně takový má i výstup. Zde autor práce vidí největší zisk, který může část systému Prestacio přinést. Grafický editor sestavovacího skriptu vytváří všechny skripty za uživatele. Díky validaci, kterou poskytuje, není možné vytvořit nevalidní sestavovací skript. Chyba se může vyskytovat pouze v samotném spuštění skriptu, kde nastane běhová chyba. Všechny běhové chyby jsou pak přehledně popsané ve výsledku sestavení. K většině chyb je v detailní popisku sepsán seznam možných příčin a jejich řešení. Tímto se snaží systém pomoci uživateli co nejrychleji odstranit chyby, které vznikly při sestavení.

6.3 Celková funkčnost a možné vylepšení

Systém byl v době vytváření testován na týmově vyvíjených aplikacích ve firmě Darkmay, s.r.o. . Díky tomu, že ho používali různí uživatelé, byly odstraněny některé základní chyby, které způsobovaly jeho nefunkčnost. Uživatelé, kteří systém používali, dodali autorovi práce několik podmětů na vylepšení systému. Systém má dobrý základ, který se může rozšiřovat. Jedno z nejdůležitějších vylepšení je rozšíření katalogů, které Hockej poskytuje. Zatím obsahuje pouze katalogy, které autor vytvořil pro splnění základní funkčnosti aplikace. Další katalogy by mohly například obsahovat komprimaci javascriptu, kontrolu syntaxe kódu nebo podporu širší množiny testů.

6.4 Systém jako volně šířitelný software

Systém bude po otestování uvolněn jako volně šířitelný software pod licencí MIT, aby jeho uživatelé mohli do systému přidávat vlastní katalogy. Tím by se systém mohl stát konkurence schopným vůči velkým komerčním systémům pro průběžnou integraci. Autor se při vytváření systému snažil klást otázky na požadavky veřejnosti na debatních fórech, které se zabývají vývojem softwaru a již zde se našlo několik dobrovolníků, kteří by se do vývoje systému chtěli zapojit.

6.5 Zhodnocení

Bylo dosaženo všech bodů zadání. Ve třetí kapitole je stručně shrnutý návrh vlastního řešení. Jeho konkrétní implementace je pak dále popsána ve čtvrté a páté kapitole práce. Systém umožňuje vykonat všechny cíle průběžné integrace, které jsou obvyklé v prostředí webových aplikací. Nicméně systém je na počátku svého vývoje a je nutné mu věnovat několik týdnů nebo možná let práce, aby se stal plnohodnotým systémem, kterým jsou již existující řešení. Autor práce bude nadále pokračovat ve vývoji systému, protože pojem průběžná integrace je nyní velmi aktuální a ze systému se může stát úspěšný projekt.

Největší úskalí při vytváření systému byla nutnost častého přepisování již existujícího kódu. Například syntaxe skriptů pro Hockej se začala stávat velice nepřehledná a bylo nutné ji od začátku změnit. Proto bylo nutné přizpůsobovat i grafický editor sestavovacích skriptů v Prestaciu.

Literatura

[1] Subversion. Open Source Software Engineering Tools [online]. 2009 [cit. 2014-05-04]. Dostupné z: http://subversion.tigris.org/

[2] AbcLinuxu. KUČERA, František. Seriál: Distribuované verzovací systémy [online]. 2011 [cit. 2014-05-04]. Dostupné z: `http://www.abclinuxu.cz/serialy/distribuovane-verzovaci-systemy`

[3] OpenStack: GitCommitMessages. Openstack [online]. 2013 [cit. 2014-04-18]. Dostupné z: `https://wiki.openstack.org/wiki/GitCommitMessages`

[4] CANNONCIAL. Bazaar [online]. 2014, 31.1.2014 [cit. 2014-05-12]. Dostupné z: http://bazaar.canonical.com/en/

[5] Gettext: manual. GNU 'gettext' utilities [online]. 2014 [cit. 2014-05-11]. Dostupné z: `http://www.gnu.org/software/gettext/manual/gettext.html`

[6] Why do we test ? What is the Purpose of Software Testing ?. In: Testing Excellence [online]. 2010 [cit. 2014-05-11]. Dostupné z: `http://www.testingexcellence.com/why-do-we-test-what-is-the-purpose-of-software-testing/`

[7] GitHub: Build software better, together [online]. 2014 [cit. 2014-05-08]. Dostupné z: `https://github.com/`

[8] Wikipedia: Comparison of continuous integration software [online]. 2014 [cit. 2014-05-08]. Dostupné z: `http://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software/`

[9] Testování software [online]. 2011 [cit. 2014-04-22]. Dostupné z: `http://testovanisoftwaru.cz/category/druhy-typy-a-kategorie-testu/`

[10] Nette framework [online]. 2014 [cit. 2014-04-22]. Dostupné z: `http://nette.org/cs/`

[11] Hosting git repositories. Gitolite [online]. 2014 [cit. 2014-05-12]. Dostupné z: http://gitolite.com/gitolite/

[12] CHACON, Scott. Pro Git. New York: Apress, c2009, xxi, 265 s. ISBN 978-1-4302-1833-3.

[13] MARTIN, Fowler. Continuous Integration. [online]. [cit. 2013-09-23]. Dostupné z: http://www.martinfowler.com/articles/continuousIntegration.html